REMARKS

The applicant submits that the present title "METHOD AND APPARATUS FOR THE DATA-DRIVEN SYNCHRONOUS PARALLEL PROCESSING OF DIGITAL DATA" is accurate. It is also consistent with the preface of the main claims (for example claim 1: "A method for data-driven synchronous parallel processing of a stream of data packets").

The applicant affirms the election of Claims 1-11 and 18-28 and has cancelled claims 12-17. The claims otherwise remain as originally filed.

The applicant notes the Examiner's comment regarding claims 1 and 28. Claim 28 in step f. defines the data token as "comprising data for associating the at least one data packet with the corresponding instruction at the one data processing unit," which claim 1 does not recite. The applicant thus submits that claim 28 is not a substantial duplicate of claim 1.

The Examiner objected to Claims 1-11 and 18-28 under 35 U.S.C. 102(b) as being clearly anticipated by Executing a Program on the MIT Tagged-Token Dataflow Architecture, 1990, IEEE, pages 300-318 (Arvind et al.). The Applicant has carefully considered the cited reference, and respectfully disagrees.

Arvind et al. is directed towards the execution of a program written in *Id*, compiled into dataflow graphs and executed on the Tagged-Token Dataflow Architecture (TTDA). The TTDA is a self-timed asynchronous multiprocessor system. The system achieves timing through the use of purely data-driven instruction scheduling. (see page 300, col. 2). The TTDA system is a data-driven processor, in that it is the arrival of data tokens, and their attendant contexts and execution rules, that serves as a trigger starting the work of each data processor. Such systems are known in the art and are differentiated from the present invention on page 3 of the subject application.

In data-driven instruction scheduling, instructions are ready to execute (referred to by Arvind et al. as 'firing') when there are tokens, input, on all of its input arcs. Tokens carry data values between operators (instructions). The execution sequence is unspecified; instructions may be executed as soon as the tokens arrive at their inputs (see page 303, col. 1). Since a parallel algorithm often has interdependencies between instructions and data elements, the system

includes contexts and execution rules, but no method for coordinating execution independent of the arrival of the data tokens. Tokens may carry a dynamic context that indicates how and when an instruction should be executed (see page 303, col. 2), but Arvind et al. does not discuss any method for synchronizing instruction execution between processors beyond the arrival of the data tokens. Arvind et al. specifically differentiates the operation of the TTDA and its reliance upon data-driven instruction scheduling, from counter-based, clock, scheduling (see page 300, col. 2).

The TTDA is a data-driven processor directed towards general-purpose computing of instructions written in the *Id* language. Timing of instruction execution is triggered by the arrival of data tokens, in consideration of their attendant contexts and execution rules. Data tokens are delivered from one processing element to the next as instructions are completed. There is no central coordinating element, clock, or arrangement of data to ensure that operations are performed in a particular order. Data is not passed between processing elements, but instead is stored in a global address space. Operations are performed as data becomes available. This is an efficient use of computing resources for applications that are not sequential or dependant upon prior results.

The present application is directed towards a synchronous microprocessor architecture. A synchronous microprocessor architecture coordinates and synchronizes allocation of resources and execution of instructions with an internal clock or timing event. The present invention uses a data-driven multiprocessor architecture to effect the parallel operation of a synchronous multiprocessor system. The invention is claimed as a method for data-driven *synchronous* parallel processing of a stream of data packets. The fact that the architecture is synchronous ensures that operations are performed in a specific order on the stream of data packets. Moreover, it ensures that the stream of data packets is maintained as a stream, since the sequential identity of the data is preserved. In contrast, the TTDA executes instructions as data-tokens arrive, but there is no facility to ensure that they arrive in a particular order. In the vector summation example on page 301 of Arvind et al., for example, the TTDA can compute the result of the summation in any order and the order or timing in which the results are available is unimportant. On the contrary, a stream of data has a specific order and operations applied to that stream may need to be applied in that order. Moreover, where sequential operations are to be

applied to segments of the data stream, both the order of operations, the order of the data elements and the state of the data elements must be preserved. These types of calculations are best performed on a synchronous microprocessor architecture.

The fact that the TTDA is asynchronous leads to inefficiencies in performing serial calculations. As noted in Arvind et al. on page 315, col. 2: "The major complication here arises from Id's non-strict semantics which makes it quite difficult to achieve efficient partitioning of code into sequential von Neumann threads." This reflects an important difference between a synchronous and an asynchronous architecture. Moreover, the data allocation described in Arvind et al. specifies that resource managers need not respond to requests in the order in which they are received. This demonstrates a key difference in the two architectures, namely that the order of operations in an asynchronous architecture, as described in Arvind et al., is related to the determinacy of the result of operations.

The various differences between the Tagged-Token Dataflow Architecture (TTDA) described in Arvind et al. and the approach taken by the present invention is explained by the fact that the former is intended for general-purpose processing (as can be seen for example in the abstract) whereas the latter is intended for processing a stream of data packets. This raises two distinct points: 1) Because the present invention works on a stream of packets, the flow of data is sequential. In general-purpose computing the flow of data can go backward, as Arvind et al. describes in his loop which causes deadlock (see page 311). 2) In the present invention the data stream is already broken into data packets, which greatly diminishes the overhead arising from the data-token approach. According to the invention a data token is attached to the entire data packet, but not just to the data for an individual instruction as required for general-purpose computing. Once again, the present invention works at he level of data, which is broken into data packets. Arvind et al. understands that working at the level of individual instructions causes intolerable overheads (see page 308) and he employs some tricks to reduce this problem, described below.

In general, the two approaches target different computational applications. Arvind et al. is suitable for general-purpose computing. The present invention is most useful for processing a stream of data packets, for example a multimedia application where one needs to decode an

MPEG-stream of digital video; other applications, for example, could be decoding of audio stream or a stream of network data.

A second important difference is that in the present invention the processing speed is regulated by data tokens of a single type – a data token showing the readiness of the packet for further processing – attached to the data packets. In Arvind et al. (probably due to the nature of general purpose computing) the data tokens regulate the computing process only locally (as he indicates at page 310). First Arvind et al. employs a special "manager" (see page 311) to regulate resources globally, then their system breaks the code into separate code blocks (see page 310) and Arvind et al. tries to achieve the data-token resource management inside each code block locally.

Here again Arvind et al. encounters two problems because of general-purpose nature of his computing environment: First, Arvind et al. has to employ a special language (Id) and its compiler to break the code into separate code blocks. Secondly, Arvind et al. must use data tokens of many different types to regulate resources inside each code block. Arvind et al. uses a "read" token, a "write" token, a "signal" token and even has to fetch data after write to check that "write" was completed (see page 307). Despite using this complicated technique Arvind et al. still encounters problems, one of which he describes as the above-mentioned deadlock loop (see page 311).

Other differences between Arvind et al. and the present invention include:

i) The present invention keeps the records of outstanding data packets requests in a special purpose fixed storage located inside the data processing unit. In Arvind et al.'s approach the "manager" allocates "I-structure," which holds the records for individual data requests, inside the so-called I-structure memory (see page 311);

ii) The control inside each processing unit is regulated purely by data tokens. Arvind et al. employs the special Control Unit inside the Processing Element (PE) that receives special tokens that can manipulate any part of the PE's state (see page 314). This is again due to the general-purpose nature of his computing, which has to function in different computing situations including a backward data flow;

iii) The multiprocessor organization of data processing in the present invention, in which each processor can perform a distinct function on a data packet. For example, one processor can perform some elements of MPEG decoding and another processor can perform different elements of MPEG decoding or even an audio decoding. In Arvind et al. each processor performs the same general-purpose computing, and he employs a sophisticated mapping mechanism to allocate code blocks across a group of processing elements (see page 314).

iv) The present invention allows for the processing of a stream of data packets in the real-time. This is important in many applications, for example multimedia applications. Arvind et al. does not have necessary mechanisms for real-time processing. In Arvind et al. the "manager" has to break code into code blocks by using compiler, which targets general-purpose computing and not real-time computing.

The invention recited in claim 1 is thus readily distinguished from the prior art. Even from the preface, Arvind et al. teach a method for general-purpose computing, not for processing a stream of data packets as claimed. These are very different computational applications, and Arvind et al.'s approach does not work for stream processing. Patentable distinctions are recited throughout the claims, for example as recited in claim 1:

Step a.: As claimed, instructions are distributed to one processing unit, before the data processing unit is available to process the instruction, whereas in Arvind et al. instructions are fetched by Instruction-Fetch Unit inside the processing element after the token has entered the Instruction-Fetch Unit (each instruction corresponding to one general-purpose operator), and all memories are globally addressed (see Multiprocessor Operation, page 314, col. 2)

Step c.: As claimed, the data requests are sent for a whole data packet, whereas in Arvind et al.'s approach the data requests are sent only for individual operands for individual general-purpose operators.

Step d.: As claimed, a record of the whole data packet requested is stored, whereas Arvind et al. stores only the records for data requests for individual operands for individual general-purpose operators.

Step e.: As claimed, the whole data packet is associated with the address of the data processing

unit which is going to process the data packet, whereas in Arvind et al. the processing units are not addressable directly (see page 314: "In multiprocessor machine all memories are globally addressed").

Step f.: As claimed, each data token is associated with a whole data packet, whereas in Arvind et al. each data token is associated with each individual operand for each individual general-purpose operator.

Step g.: As claimed, the whole arriving data packet is associated with an instruction, whereas Arvind et al. associates each arriving data unit with either one operand of a monadic individual general-purpose operator or one of two operands of a dyadic individual general-purpose operator (see page 314).

Step h.: As claimed, the whole data packet is processed according to one instruction, whereas Arvind et al. executes an individual general-purpose operator when all operands have arrived.

These distinctions also apply to claim 28.

Similarly, regarding the Examiner's objection to claim 18, again because Arvind et al. teaches an apparatus for general-purpose computing, but not data-driven synchronous processing of data packets, in the present invention each digital data processor contains multiple data processing units whereas in Arvind et al. multiple processing elements are directly connected together via network. Distinctions from Arvind et al. include:

-- As claimed, a digital data processor receives and sends data and instruction from/to external devices and distributes them to multiple data processing units, whereas in Arvind et al. multiple processing elements are connected together through a network.

-- As claimed, the digital data processor has a separate Instruction Path and Data Path, whereas in Arvind et al. instruction and data paths are mixed together in the Main Pipeline inside each processing element (see page 314).

-- As claimed, the digital data processor has a plurality of data processing units organized for parallel processing, whereas in Arvind et al. multiple processing elements are directly connected.

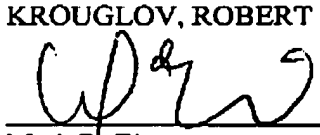
-- As claimed, the instruction-distributing unit is located inside a digital data processor and distributes instructions to multiple data processing units, whereas in Arvind et al. each Instruction Fetch Unit is located inside each processing element (PE) itself.

The remaining claims are dependent, directly or indirectly, on these main claims, and are therefore similarly allowable.

The Applicant accordingly submits that Arvind et al. does not anticipate the claims as originally filed in the present application. Favourable reconsideration and allowance of this application are respectfully requested.

Executed at Toronto, Ontario, Canada, on October 26, 2004.

DANIEL GUDMUNSON, ALEXEI KROUGLOV, ROBERT COLEMAN



Mark B. Eisen
Registration No. 33,088
(416) 971-7202, Ext. 242
Customer Number: 38735

MBE:lf
Encl. Replacement Figures 1-5